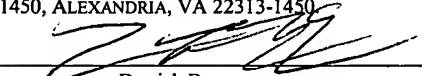


PATENT
5760-20100
VRTS-0691

"EXPRESS MAIL" MAILING LABEL NUMBER
EL990143733US

DATE OF DEPOSIT FEBRUARY 27, 2004

I HEREBY CERTIFY THAT THIS PAPER OR
FEE IS BEING DEPOSITED WITH THE
UNITED STATES POSTAL SERVICE
"EXPRESS MAIL POST OFFICE TO
ADDRESSEE" SERVICE UNDER 37 C.F.R.
§1.10 ON THE DATE INDICATED ABOVE
AND IS ADDRESSED TO THE
COMMISSIONER FOR PATENTS, P.O. Box
1450, ALEXANDRIA, VA 22313-1450.


Derrick Brown

INTERNAL MONITORING OF APPLICATIONS
IN A DISTRIBUTED MANAGEMENT FRAMEWORK

By:

Michael P. Spertus

Christopher D. Metcalf

Richard Schooler

David A. Stuckmann

BACKGROUND OF THE INVENTION

Field of the Invention

5 **[0001]** This invention is related to the field of application performance management and, more particularly, to internal monitoring of applications in a distributed management framework.

Description of the Related Art

10

[0002] In the information technology (IT) departments of modern organizations, one of the biggest challenges is meeting the increasingly demanding service levels required by users. With more and more applications directly accessible to customers via automated interfaces such as the world wide web, "normal" business hours for many enterprises are
15 now 24 hours a day, 7 days a week. The need for continuous availability and performance of applications has created complex, tiered IT infrastructures which often include web servers, middleware, networking, database, and storage components. These components may be from different vendors and may reside on different computing platforms. A problem with any of these components can impact the performance of
20 applications throughout the enterprise.

[0003] The performance of key applications is a function of how well the infrastructure components work in concert with each other to deliver services. With the growing complexity of heterogeneous IT environments, however, the source of performance
25 problems is often unclear. Consequently, application performance problems can be difficult to detect and correct. Furthermore, tracking application performance manually can be an expensive and labor-intensive task. Therefore, it is usually desirable that application performance management tasks be automated.

[0004] Automated tools for application performance management may assist in providing a consistently high level of performance and availability. These automated tools may result in lower costs per transaction while maximizing and leveraging the resources that have already been spent on the application delivery infrastructure. Automated tools for application performance management may give finer control of applications to IT departments. Application performance management tools may enable IT departments to be proactive and fix application performance issues before the issues affect users. Historical performance data collected by these tools can be used for reports, trending analyses, and capacity planning. By correlating this collected information across application tiers, application performance management tools may provide actionable advice to help IT departments solve current and potential problems.

[0005] Typically, a significant portion of any IT budget is incurred in trying to sustain business objectives. Without the ability to peer into the quality of application software, the benefits of other IT investments cannot easily be seen, and efforts spent on ensuring optimal performance of infrastructure components may never be appreciated. IT organizations are often required to manage deployment of business-critical applications that have poor reliability, performance, or scalability. Server-based enterprise applications present a particularly significant challenge for IT organizations trying to provide required service levels while controlling IT infrastructure costs. It is desirable to provide improved systems and methods for application performance management, health monitoring, and error resolution.

SUMMARY OF THE INVENTION

[0006] Various embodiments of systems and methods described herein may provide internal monitoring of applications in a distributed management framework. The distributed management framework may comprise a plurality of applications and application servers. In one embodiment, each of the applications is configured to make function calls to standard programming functions. The function calls to the standard programming functions may be intercepted and routed to alternative implementations of the standard programming functions. Alternative implementations of the standard programming functions may be used to collect availability metrics for the applications.

[0007] In one embodiment, a manager thread may be started inside each of the plurality of applications. The manager threads may be used to monitor execution of the plurality of applications in a production environment. For example, the manager threads may be used to detect hung processes.

[0008] In one embodiment, the program code of a monitored application may be modified to include additional instructions. The additional instructions may be used to monitor execution of the monitored application in a production environment. In response to a triggering event in the execution of the monitored application, output may be automatically generated. The output may comprise an execution history for the monitored application.

BRIEF DESCRIPTION OF THE DRAWINGS

[0009] Figure 1 illustrates components of a performance management system including internal monitoring of applications according to one embodiment.

5

[0010] Figure 2 illustrates components of an exemplary computer system with which embodiments of a system and method for performance management may be implemented.

[0011] Figure 3A is a table which shows metrics for the monitoring of user application activity according to one embodiment.

10

[0012] Figure 3B is a table which shows metrics for the detection and/or correction of user application errors according to one embodiment.

[0013] Figure 3C is a table which shows system metrics related to internal monitoring of applications according to one embodiment.

15

[0014] Figure 3D is a table which shows metrics for other runtime agent activity according to one embodiment.

20

[0015] Figure 3E is a table which shows derived metrics related to internal monitoring of applications according to one embodiment.

[0016] Figure 4 is a flowchart which illustrates a method for internal monitoring of applications using function call interception according to one embodiment.

25

[0017] Figure 5A is a table which shows instrumentation options according to one embodiment.

[0018] Figure 5B is a table which shows other options related to instrumentation according to one embodiment.

5 [0019] Figure 6 is a flowchart which illustrates a method for internal monitoring of applications using additional internal program code according to one embodiment.

[0020] Figure 7 is a flowchart which illustrates a method for internal monitoring of applications using a manager thread according to one embodiment.

10 [0021] Figure 8 illustrates a user interface for internal monitoring of applications including a Dashboard workspace according to one embodiment.

[0022] Figure 9 illustrates a user interface for internal monitoring of applications including a Health workspace according to one embodiment.

15

[0023] Figure 10 illustrates a user interface for internal monitoring of applications including an Activity workspace according to one embodiment.

20 [0024] Figure 11 illustrates a user interface for internal monitoring of applications including a Forensics workspace according to one embodiment.

25 [0025] While the invention is described herein by way of example for several embodiments and illustrative drawings, those skilled in the art will recognize that the invention is not limited to the embodiments or drawings described. It should be understood, that the drawings and detailed description thereto are not intended to limit the invention to the particular form disclosed, but on the contrary, the intention is to cover all modifications, equivalents and alternatives falling within the spirit and scope of the present invention as defined by the appended claims. As used throughout this application, the word "may" is used in a permissive sense (i.e., meaning "having the

potential to"), rather than the mandatory sense (i.e., meaning "must"). Similarly, the words "include," "including," and "includes" mean "including, but not limited to."

DETAILED DESCRIPTION OF EMBODIMENTS

[0026] In one embodiment, a performance management system may include internal monitoring of applications for management of distributed enterprise applications. In one
5 embodiment, the performance management system may provide monitoring of application reliability, performance, and scalability through a console that reports on problems. In one embodiment, the performance management system may provide solutions to common software problems with tuning solutions that prevent software faults from becoming application failures. In one embodiment, the performance management
10 system may provide root-cause analysis through detailed forensic snapshots that can reconstruct the chain of events leading up to an application fault or failure.

[0027] Figure 1 illustrates components of a performance management system 100 including internal monitoring of applications according to one embodiment. In one
15 embodiment, the performance management system 100 may enable operators and administrators to manage and improve the scalability and reliability of their existing application software through visualization of an application's health and dynamic optimization and remediation of software bottlenecks and errors. To monitor usage of dynamic resources and provide facilities for enhancing the behavior of application
20 software, a Runtime Agent 332 may intercept many of the well-defined APIs used by programmers in the implementation of application programs. The Runtime Agent 332 may be integrated with application programs using in-process instrumentation of the application image.

25 [0028] By modifying the in-process image of an application rather than the file stored on disk, the performance management system is minimally invasive. By using interception techniques in place of debugging modes (e.g., the Unix ptrace service), the performance management system may be used to diagnose, optimize, or remediate problems without the severe performance overhead or security risks that are typically associated with
30 debuggers and profilers. In one embodiment, installation and integration of the Runtime

Agent 332 and its associated components may be accomplished by an Administrator without developer intervention.

5 [0029] In one embodiment, the Runtime Agent 332 runs in application programs to monitor and enhance their health. The Runtime Agent 332 may insert probes into the applications and intercept calls into the system infrastructure to allow monitoring and enhancement. Runtime Agents 332 may be implemented as shared or dynamically-loadable libraries. Different agents may be provided for different platforms and underlying programming languages. In one embodiment, the Runtime Agent 332 is
10 implemented as a set of DLLs (e.g., on computer systems using a Microsoft Windows® OS). In one embodiment, the Runtime Agent 332 is implemented as a set of shared libraries (e.g., on Unix-based computer systems). These libraries run inside the application process, thereby bypassing the overhead associated with traditional, out-of-process monitoring tools. In general, the appropriate Runtime Agents 332 are
15 automatically selected, without the need for user knowledge of this structure.

[0030] In one embodiment, a Host Collector 334 process runs on each monitored server 330 and collects metrics, alerts, and self-correction updates from the Runtime Agents 332 for the various monitored applications on the host 330. The Host Collector 334 may add
20 additional per-host metrics and periodically provide the collected information to a FocalPoint Server 320.

[0031] In one embodiment, the FocalPoint Server 320 provides a Console 312 with metrics, health, and correction and forensics information. The FocalPoint Server 320
25 may also archive health information to provide for the viewing of data over time in the Console 312.

[0032] In one embodiment, the Console 312 is an operational interface for the performance management system. The Console 312 may be used to view metrics,
30 monitor application health, and review recommended corrections. The Console 312 may

be used to depict metrics and other information in overall summary form, with a user-selected time-frame. The Console 312 may be used to drill down to module level to identify specific custom or vendor libraries. The Console 312 may be used to change product configurations, intervene with or re-configure user applications, and display alerts. An alert is an indication of an exceptional event. Alerts may be discovered by the performance management system itself, and the performance management system be configured by the user to be triggered by metrics exceeding thresholds. In one embodiment, an Agent Installer 314 distributes and tracks the performance management system components across the distributed installation. The Console 312 and Agent Installer 314 may reside on an Installation Server 310.

[0033] In one embodiment, a Listener 336 may run on each application server 330 where the performance management system components are installed. The Listener 336 may handle install and upgrade requests and transfer data for the Host Collector 334. In one embodiment, a Listener 336 may be installed temporarily on the FocalPoint Server 320 but deleted during the installation.

[0034] In one embodiment, a component called Forensics 338 may optionally capture relevant system information and the exact sequence of statement execution across all threads when an application fails or at other times specified by the user. Users may view forensics files using a Forensics Viewer.

[0035] An EndPoint 330 is an application server on which various components such as a Runtime Agent 332, Host Collector 334, Listener 336, and, optionally, Forensics 338 have been installed. For purposes of illustration, Figure 3 depicts two EndPoints 330 and one FocalPoint 320. However, other numbers and combinations of EndPoints and FocalPoints may be used in the performance management system.

[0036] The performance management system 100 of Figure 1 may be executed by one or more networked computer systems. The various components of the performance

management system 100 may reside on the same computer system, on different computer systems, or on an arbitrary combination of computer systems. Figure 2 is an exemplary block diagram of such a computer system 200. The computer system 200 includes a processor 210 and a memory 220 coupled together by communications bus 205. The processor 210 can be a single processor or a number of individual processors working together. The memory 220 is typically random access memory (RAM), or some other dynamic storage device, and is capable of storing instructions to be executed by the processor 210. For example, the instructions may include instructions for the performance management system 100. The memory 220 may store temporary variables or other intermediate information during the execution of instructions by the processor 210. The memory 220 may store operating system (OS) software to be executed by the processor 210.

[0037] In various configurations, the computer system 200 may include devices and components such as a keyboard & mouse 250, a SCSI interface 252, a network interface 254, a graphics & display device 256, a hard disk 258, and/or a CD-ROM 260, all of which are coupled to the processor 210 by a communications bus 207. The network interface 254 may provide a communications link to one or more other computer systems via a LAN (local area network), WAN (wide area network), internet, intranet, or other appropriate networks. It will be apparent to those having ordinary skill in the art that the computer system 200 can also include numerous elements not shown in the figure, such as additional storage devices, communications devices, input devices, and output devices, as illustrated by the ellipsis.

[0038] In one embodiment, the environment is the set of applications and hosts that make up the user's sphere of interest, which we model as the following hierarchy: Environment (user-defined), Application (user-defined), Process (system-defined, running on a single Host), Component (user-defined), Module (system-defined), Code (symbol, stack-trace, source). In one embodiment, the environment may be referred to within the performance management system as "All Applications." An application is a software solution, such as

various Commercial Off-The Shelf (COTS) products or custom applications. An application may contain multiple executables, shared objects, and other suitable components. An application instance is a running application. The application instance may include multiple processes running on multiple CPUs or computer systems. As
5 defined within an environment, a process is an operating system process or task, typically comprising a main executable and several dynamically-loaded or shared libraries. As defined within an environment, a component is a logical part of a process, generally including one or more logically related modules in a process. As defined within an environment, a module is a binary element of a process or component, such as executable
10 or shared library, Java class, or J2EE component such as an EJB.

[0039] The Runtime Agents 332 may collect availability metrics for application programs running on the monitored application servers 330. Availability metrics may include, for example, metrics related to memory errors, heap fragmentation, lock contentions, signals,
15 and hung process detection. The FocalPoint 320 may provide higher-level aggregation of metrics.

[0040] In one embodiment, the Runtime Agents 332 may collect the metrics listed in Figures 3A – 3E. Figure 3A is a table which shows metrics for the monitoring of user
20 application activity according to one embodiment. Figure 3B is a table which shows metrics for the detection and/or correction of user application errors according to one embodiment. Figure 3C is a table which shows system metrics according to one embodiment. Figure 3D is a table which shows metrics for other runtime agent activity according to one embodiment. Figure 3E is a table which shows derived metrics
25 according to one embodiment. The derived metrics shown in Figure 3E may be created using combinations of the base metrics shown in Figures 3A through 3D.

[0041] In one embodiment, any of the listed metrics may be displayed either as current
30 values or graphed over time. In one embodiment, each of the metrics listed in Table 1 may have the following configuration options: enable on/off; threshold that denotes

“warning” level (e.g., depicted in yellow in Console); and threshold that denotes “error” level (e.g., depicted in red in Console). For some metrics, the threshold may be a single such event, for example a hung process or an abnormally terminated process. In general, the metrics listed in Figures 3A – 3E are available across all supported operating environments. One exception is Java, for which some metrics, particularly memory errors, may not be applicable.

[0042] Figure 4 is a flowchart which illustrates a method for internal monitoring of applications according to one embodiment. In 402, agents (e.g., Runtime Agents 332) are inserted into a plurality of applications on a monitored computer system (e.g., an application server 330). In one embodiment, an agent is inserted into an application process when the application is launched. The application images in memory, and not the applications as stored on disk, may be modified by the insertion of the agents.

[0043] Each of the applications is configured to make function calls to standard programming functions. The standard programming functions may include, for example, functions provided by a particular programming language (e.g., memory functions such as malloc, free, new, delete, etc. in a C/C++ environment) and OS-specific or platform-specific functions (e.g., HeapAlloc functions in a Windows® environment). In 404, function calls to the standard programming functions are intercepted. In one embodiment, the agents are configured to perform the interception of the function calls. The function calls may be intercepted in a production environment as well as in a development environment, testing environment, or other suitable environment. As used herein, the term "production environment" is a computing environment in which end users may use an application to accomplish day-to-day functions. By contrast, the term "development environment" is a computing environment in which developers (e.g., programmers) may develop and/or test unfinished applications. A production environment includes deployed applications, not applications that are under development.

[0044] In 406, the function calls are routed to alternative implementations of the standard programming functions instead of the standard implementations. The alternative implementations may comprise libraries or other shared resources that replace the functionality of standard libraries and OS functions. In 408, the alternative
5 implementations of the standard programming functions may be used to collect availability metrics for the one or more applications. The alternative implementations may reside inside the application processes, where they monitor resources such as memory usage.

10 [0045] The availability metrics may then be used by the performance management system to analyze application health and resource usage, make recommendations, and implement corrective actions. In one embodiment, the performance management system may comprise a distributed management framework as described with reference to Figure 1. The distributed management framework may comprise various combinations of
15 applications on various servers and platforms. The distributed management framework may be configured by an Administrator to monitor a subset of the applications and/or servers using the method shown in Figure 4. These various combinations of applications may then be managed using the Console 312.

20 [0046] In one embodiment, a method referred to as injection may be used by the Administrator to insert the Runtime Agent 332 into a specific application. Injection may take place through interaction with the server's program loader. In one embodiment, injection may be enabled for applications (e.g., C/C++ applications) in a Windows® environment by modifying a central configuration file. The configuration file may be
25 edited by adding strings for the executables to be injected:

native.runtime.Inject=<executable>;<executable>;...

[0047] In one embodiment, each term <executable> can be an executable name (e.g., myapp.exe) or an executable name and path (e.g., C:\Program Files\myapp.exe). In one

embodiment, applications will not be injected if the host is booted in "Safe Mode." (e.g., in Windows® 2000 and higher).

5 [0048] In one embodiment, e.g., on Unix-based systems, injection is part of the way an executable is invoked. Typically in such environments, an executable is invoked as a command with arguments. To invoke the executable with injection, a command such as "gsinject" may be used to launch the executable. The administrator should decide whether child processes launched by the executable should also be injected. In one embodiment, forked processes are always injected. If a process is forked and then execed,
10 it will only be injected if the -e flag is used in one embodiment. In one embodiment, the syntax of the injection command "gsinject" is as follows:

gsinject [-n|e] [-i bytes] [-v] [-h] [-c license-file] command [arguments]

[0049] If the option -n is used, child processes launched using exec will not be injected.
15 In one embodiment, -n is the default. If the option -e is used, child processes launched via exec and fork will be injected. If the option -i is used, the initial heap size will be increased by the specified number of bytes. If the option -v is used, messages about injection will be emitted to standard output. If the option -h is used, help text for the command will be displayed. If the option -c is used, the location of the text license file
20 may be specified, and any default locations or variables overridden. For example, to inject into myprog and all child processes, the following command may be used:

gsinject -e myprog arg1 arg2

[0050] In addition to injection, instrumentation may be used for J2EE™ and Java
25 programs. Instrumentation inserts probes in Java and native programs. Instrumentation may also be used to insert probes that generate forensic reports for both Java and native programs. Instrumentation may use compiler technology to perform binary instrumentation on executables, DLLs, shared libraries, J2EE Enterprise Java Beans, Java archives, and other suitable software components.

30

[0051] In one embodiment, the performance management system may provide two ways to instrument Java code: statically and dynamically. Static instrumentation is a manual command line operation that allows the administrator to instrument class and archive files individually. Dynamic instrumentation allows the administrator to automatically
5 instrument a Java program upon launching it.

[0052] For static instrumentation in one embodiment, the Java code is compiled into bytecodes before it is instrumented. In one embodiment, a Java application may be instrumented using a command such as "tfj-instr." In one embodiment, this command
10 will enable only metrics (and not forensics) by default; one or the other or both may be specified using command line options. The Java program may then be executed in the normal way, provided that a performance management system runtime JAR should be available. To make tfj-runtime.jar available, tfj-runtime.jar should be placed in the JRE's extension directory and also on the classpath. To put it on the classpath, either set or add
15 the CLASSPATH environment variable or set or add to the -classpath Java command-line option (often abbreviated as -cp).

[0053] Figure 5A is a table which shows instrumentation options for an instrumentation command according to one embodiment. Figure 5B is a table which shows other
20 command-line options for an instrumentation command according to one embodiment. In one embodiment, the tfj-instr command may have the following syntax, where the options are illustrated in Figures 5A and 5B:

```
tfj-instr [-v] [-h] [-metrics|forensics] [-full] [-q] [-qi] [-s] [-no-manifest-update] [-d  
  <out-dir>] [-label <label>] [-log <log-file>] [-p <policy.opt>] [-j <map-file.jar>]  
25  [-sourcepath <source-path>] [-batch <batch-file>] [-bs <backup-suffix>] [-ts  
  <MMddyyyyHH:mm:ss>] [-tf <MMddyyyyHH:mm:ss>] {-r <dir>} *.class ... *.jar  
  ...
```

[0054] In one embodiment, J2EE components may be instrumented and deployed. First,
30 the classes to be traced may be instrumented. Second, the forensics runtime JAR (tfj-runtime.jar) may be placed in the extensions directory of the Java Virtual Machine (JVM)

that will be running the application server 330. If this is not possible, the runtime JAR should at least be in the classpath of the executing application server 330.

[0055] For dynamic instrumentation, a class loader may be used to automatically launch
5 the Java program and instrument all associated class files. To use the dynamic
instrumentation tool, the class file should be compiled first. Second, the following
command should be entered: `tfj-ijava <ClientClass>`, where `<ClientClass>` is the name of
the class file which the user wants to instrument and open. To dynamically instrument
the class without using the `tfj-ijava` script, the following command should be entered,
10 with a `CLASSPATH` setting that includes `tfj-runtime.jar`: `java`
`incert.instrument.Launcher <ClientClass>`. In one embodiment, all class files contained
within `<ClientClass>` are instrumented automatically while they execute. Map (.tj1) files
may be saved into a directory hierarchy. Whereas static instrumentation produces a new
set of class files, dynamic instrumentation may eliminate the need for maintaining both an
15 uninstrumented and instrumented code base in a development environment.

[0056] In one embodiment, the Forensics Agent 338 may identify the root cause of
failures in applications (e.g., C/C++ applications) in development, testing, and
production. By monitoring program execution and system information, the Forensics
20 Agent 338 may pinpoint the exact source of the failure, thereby helping the administrator
to rapidly recover from the problem, minimizing debugging time, and eliminating the
need to re-create problems. Figure 6 is a flowchart which illustrates a method for internal
monitoring of applications using additional internal program code according to one
embodiment. The program code of a monitored application may be modified to include
25 additional instructions in 602. As used herein, the term "program code" includes machine
code in addition to other types of instructions which may be executed or interpreted
without an additional compilation step. In one embodiment, for example, the Forensics
Agent 338 may automatically insert lightweight agents into program modules (e.g.,
C/C++ modules) and Java bytecode through instrumentation.

30

5 [0057] As the instrumented program runs, the Forensics Agent 338 continuously monitors its execution in 604. The Forensics Agent 338 may capture the exact sequence of statement execution across all threads and relevant system information. When a triggering event such as an exception, error, or termination occurs in the instrumented application, the Forensics Agent 338 automatically generates and outputs forensics files that capture the execution history of the target program in 606. The execution history may comprise a history of source statement execution. The forensics files may then be analyzed to identify the cause of the failure and quickly fix the problem without needing to re-create the failure in a test environment. In one embodiment, forensics files may also be generated through an external command at an arbitrary time.

15 [0058] In one embodiment, the Forensics Agent 338 may add additional instructions to the program code of the target application. The additional instructions may be used to record relevant aspects of the program's execution in a production environment. In one embodiment, the additional instructions may record an execution trace of the program's execution on a per-thread basis. In one embodiment, the additional instructions may record entry to and exit from function calls executed during program execution. In one embodiment, the additional instructions may capture exceptional control transfers and record details of the associated exceptions. In one embodiment, the additional instructions may track the creation of specific data objects and record metrics for the creation events.

25 [0059] In one embodiment, (e.g., a Windows® environment), a program may be compiled in the usual manner for use with the Forensics Agent 338. Any program files (e.g., EXE, DLL, and OCX files) may then be instrumented by inserting agents into the program files. New, instrumented versions of the program files are thereby created. When the instrumented modules are executed, forensics files may be generated when exceptions, errors, or abnormal terminations occur. In one embodiment, forensics files may also be generated manually at any time.

30

[0060] In one embodiment, (e.g., a Solaris® environment), a program may be compiled for use with the Forensics Agent 338 with debugging enabled, and a forensics wrapper program may be used during the link process to instrument the code. When the instrumented modules are executed, forensics files may be generated when signals, errors, or abnormal terminations occur. In one embodiment, forensics files may also be generated manually at any time.

[0061] In one embodiment (e.g., a Java® environment), a class and/or archive may be instrumented for use with the Forensics Agent 338. Forensics files may be generated when the instrumented code is executed.

[0062] Figure 7 is a flowchart which illustrates a method for internal monitoring of applications using a manager thread according to one embodiment. A manager thread may be started inside each of the plurality of applications in 702. The manager threads may be used to monitor execution of the plurality of applications in a production environment in 704. For example, a manager thread inside a particular process may be used to determine that the particular process is hung. If the manager thread does not receive sufficient time to perform its activities and keep in contact with the host collector, the process can be considered hung. This internal method of hung process detection is typically more accurate than the normal method, i.e., testing whether an application responds to external stimuli. In one embodiment, after collecting availability metrics for the applications using internal monitoring, the manager threads may be used to trigger output of the availability metrics to an external recipient (e.g., a FocalPoint or Console). In one embodiment, manager threads may be used to trigger garbage collection of memory resources.

[0063] Figures 8 – 11 are screenshots which illustrate a user interface for internal monitoring of applications according to one embodiment. In one embodiment, the Console 312 provides four main workspaces. In one embodiment, an individual workspace may be selected from the Console 312 by selecting the appropriate tab. Figure

8 illustrates an example of a Dashboard workspace according to one embodiment. The Dashboard workspace may provide an overview of all applications, current health status and self-corrections. Health alerts are displayed by type and in overview. Figure 9 illustrates an example of a Health workspace according to one embodiment. The Health workspace may include an event viewer area that provides a list of alerts, self-corrections, details on the selected event, and recommendations where appropriate. Figure 10 illustrates an example of an Activity workspace according to one embodiment. The Activity workspace may provide a main metrics-display area. The metrics may be viewed by all applications or by drill-down through the overall environment. Figure 11 illustrates an example of a Forensics workspace according to one embodiment. The Forensics workspace may list further data, such as forensic snapshot files.

[0064] It is further noted that any of the embodiments described above may further include receiving, sending or storing instructions and/or data that implement the operations described above in conjunction with Figs. 1 – 11 upon a computer readable medium (also referred to herein as a "carrier medium"). Generally speaking, a computer readable medium may include storage media or memory media such as magnetic or optical media, e.g. disk or CD-ROM, volatile or non-volatile media such as RAM (e.g. SDRAM, DDR SDRAM, RDRAM, SRAM, etc.), ROM, etc. as well as transmission media or signals such as electrical, electromagnetic, or digital signals conveyed via a communication medium such as network and/or a wireless link.

[0065] Although the embodiments above have been described in considerable detail, numerous variations and modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated. It is intended that the following claims be interpreted to embrace all such variations and modifications.